Learning Outcomes: 1) understanding more about logic and gates; 2) understanding truth tables; 3) getting used to assembly-level thinking; 4) looking at code fragments in assembly without writing an entire program

1. Consider the function with three inputs (A,B,C) and two outputs (X,Y) that works like this:

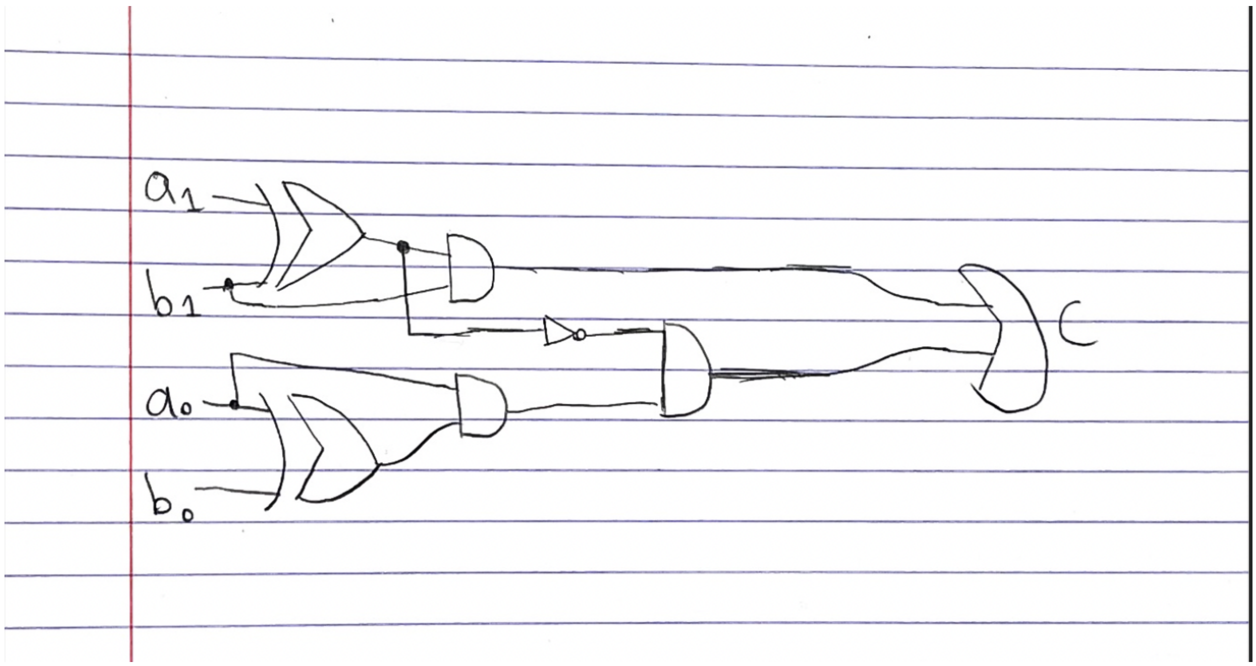
```
 A  B  C | X  Y
---------+------
 0  0  0 | 0  1
 0  0  1 | 0  1
 0  1  0 | 0  1
 0  1  1 | 1  1
 1  0  0 | 1  0
 1  0  1 | 1  1
 1  1  0 | 1  0
 1  1  1 | 1  1
```

Design two logic circuits for this function, one using AND, OR and NOT gates only, and one using NAND gates only. *You DO NOT HAVE to draw the circuit*, but it might be helpful to do that to visualize and trace the logic. However, for this question you are only required to write the two formulas — one for computing **X** and one for computing **Y**. They can take the form of a logical equation such as

X := A and B or such as **Y := not-B and (A or C)**.

a) **X := A or (B and C), Y := (not-A or C)**
b) **X := (A NAND B) NAND (B NAND C), Y := A NAND (A NAND C)**

2. Draw a logic circuit that compares two 2-bit signed numbers as follows. It should have four inputs a1, a0, b1, and b0. a1a0 is a 2-bit signed number (call it a) and b1b0 is a 2-bit signed number (call it b). The circuit has one output, c, which is 1 if a > b and 0 otherwise.



3. Given a 32-bit register, write logic instructions to perform the following operations. For parts (c) and (f) assume an unsigned interpretation; for part (d) assume a signed interpretation.

a. Clear all even numbered bits.

Logic Operator: AND, Mask: 0xAAAAAAAA

AND 1010 1010 1010 1010 1010 1010 1010 1010 (in binary)

b. Set the last three bits.

Logic Operator: OR, Mask: 0x00000007

OR 0000 0000 0000 0000 0000 0000 0000 0111

c. Compute the remainder when divided by 8.

(Greatest possible remainder: 7 or 111. Leave these bits be.)

Logic Operator: AND   Mask: 0x00000007

AND 0000 0000 0000 0000 0000 0000 0000 0111

d. Make the value -1

Logic Operator: OR, Mask: 0xFFFFFFFF

OR 1111 1111 1111 1111 1111 1111 1111 1111

e. Complement the two highest order bits

Logic Operator: XOR, Mask: 0xC0000000

XOR 1100 0000 0000 0000 0000 0000 0000 0000

f. Compute the largest multiple of 8 less than or equal to the value itself

(Opposite of C! For example 23 // 8 = 2 r7. Clear remainder.)

Logic Operator: AND, Mask: 0xFFFFFFF8

AND 1111 1111 1111 1111 1111 1111 1111 1000

4. For the sample single-accumulator computer discussed in class, write a complete assembly language program in the **stanley/penguin** language that sends the values **0** through **255** out to port **0x8**. NOTE: the machine code for this will be written in the next problem.

```
FUNCTION NAME: output0to255
AUTHORS: adi, jd, nolan

Fxn description:
    Writes out to port 0x8
    vars:
        - value(to be written out, starts at 0)
        - increment(increases value by 1)
        - max(set to 256)

                JMP         start
value:          0
increment:      1
max:            256
start:          LOAD        value
                WRITE       8
                ADD         increment
                STORE       value
                SUB         max
                JLZ         start
end:            JUMP        end
```

5. Translate your assembly language program in the previous problem to machine language.

```
FUNCTION NAME: output0to255 (with machine code!)
AUTHORS: adi, jd, nolan

Fxn description:
    Writes out to port 0x8
    vars:
        - value(to be written out, starts at 0)
        - increment(increases value by 1)
        - max(set to 256)

C0000004    JMP         start
00000000    value:          0
00000001    increment:      1
00000100    max:            256
00000001    start:          LOAD        value
30000008                    WRITE       8
40000002                    ADD         increment
10000001                    STORE       value
50000003                    SUB         max
E0000004                    JLZ         start
C000000A    end:            JUMP        end
```

6. For the sample single-accumulator computer discussed in class, write a complete assembly language program in the **stanley/penguin** language that computes a greatest common divisor. Assume the two inputs are read in from port **0x100**. Write the result to port **0x200**. You do not need to write machine code for this problem.

```
FUNCTION NAME: greatestCommonFactor
AUTHORS: adi, jd, nolan

Fxn description:
    Reads two values from a port, and calculates their greatest common factor.

                JMP             start
val1:           0
val2:           0
factor:         0
gcf:            1
start:          READ            100             ; Storing val1 and val2.
                STORE           val1
                READ            100
                STORE           val2
                SUB             val1            ; If they're equal, that's the GCF.
                JZ              equalcase

newfactor:      LOAD            factor          ; Incrementing factor.
                ADD             1
                STORE           factor
                LOAD            val1
                MOD             factor
                JGZ             newfactor       ; If val1 isn't divisible, increment again.
                LOAD            val2
                MOD             factor
                JGZ             newfactor       ; If val1 is divisible but val2 isn't, increment again.
                LOAD            factor
                STORE           gcf             ; If both are divible by the factor, store that for now.
                LOAD            val1
                SUB             factor          ; If the factor is equal to val1, then
                JZ              done            ; this is the largest factor we can have.
                LOAD            val2
                SUB             factor          ; If the factor is smaller than val2 but
                JZ              done            ; equal to val2, same thing.
                JMP             newfactor

equalcase:      LOAD            val1            ; If the values are equivalent, their value is the
                WRITE           200             ; Greatest Common Factor. Write it to 0x200 and end.
                JMP             end

done:           LOAD            gcf             ; Once we max out on possible factors, we load the greatest
                WRITE           200             ; GCF we found and write it out.

end:            JMP             end
```

7. For the sample single-accumulator computer discussed in class, give a code fragment, in assembly language of the **stanley/penguin** language, that swaps the accumulator and memory address **0x30AA**. You do not need to write machine code for this problem.

```
FUNCTION NAME: swapValues
AUTHORS: adi, jd, nolan

Fxn description:
    Swaps the accumulator value and memory address 0x30AA.

temp1:          0
temp2:          0
                STORE           temp1       ; Storing Acc value in temp1.
                LOAD            0x30AA      ;
                STORE           temp2       ; Storing 0x30AA value in temp2.
                LOAD            temp1
                STORE           0x30AA
                LOAD            temp2
```

8. For the sample single-accumulator computer discussed in class, give a code fragment, in assembly language of the **stanley/penguin** language that has the effect of jumping to the code at address **0x837BBE1** if the value in the accumulator is greater than or equal to **0**. You do not need to write machine code for this problem.

JGZ 837BBE1

JZ 837BBE1

9. **Part 1 of 2**: Explain, at a high-level, what the following sequence of instructions does. In other words, suppose a programmer has stored data in **r8** and **r9**. After executing these instructions, what does the programmer notice about the data?

```
xor r8, r9
xor r9, r8
xor r8, r9
```

The programmer will notice that the values of the data will have switched between r8 and r9.

Part 2 of 2: Also state as briefly as possible why that effect happens.

This happens because when we use a logic operator that compares two values, its result will be a bit sequence of the results of applying the operator to both values. That sequence of bits makes up a value, and will replace the first value that's being compared.

In this case, when we **xor** r8 and r9, assuming that they are both values that can be shown as a sequence of bits, a new sequence of bits will be generated which would be r8's new value. After following up with **xor r9, r8** and **xor r8, r9**, the end result just happens to be switching the two values.